APPLE II UTILITY PACK

By Roger Wagner

Now all the routines most used by programmers are in one convenient set! A real software bargain with all 11 programs in one package at less than $2.00 per program! Here's what you get:

RENUMBERING — INTEGER & APPLESOFT	Includes special features like printer output of old/new line #'s for later reference, and warns of unusual conditions, like 'GOTO A*10' in Integer. This is one of the few renumber routines that is **non-destructive to machine language code** contained in Applesoft listings!
APPEND — INTEGER & APPLESOFT	Easily join program modules together.
ADDRESS/HEX CONVERTER	Converts all of the Apple's address formats including high- and low-order bytes for pointers.
LINE FIND — INTEGER & APPLESOFT	Find the location of any BASIC line in memory for repairing garbaged programs, or inserting HIMEM:, CLR, etc. directly into a listing.
VAL( ) & STR$( )	Instructions for simulating these Applesoft functions in Integer BASIC. Convert strings to numbers and back again with these.
SCREEN PAGE MAP	Put any character anywhere on the screen by 'POKE'ing directly into memory the proper values, given by this program.
MOVE	Move blocks of memory up or down any number of bytes from Integer BASIC or Applesoft.

The extensive documentation alone is worth the price! It includes an in-depth explanation of the internal workings of Integer BASIC and Applesoft to allow you to do things you never thought possible on your APPLE!

Southwestern Data Systems
P.O. Box 582
Santee, CA 92071
(714) 562-3670

APPLE II UTILITY PACK

By Roger Wagner

Now all the routines most used by programmers are in one convenient set! A real software bargain with all 11 programs in one package at less than $2.00 per program! Here's what you get:

RENUMBERING — INTEGER & APPLESOFT	Includes special features like printer output of old/new line #'s for later reference, and warns of unusual conditions, like 'GOTO A*10' in Integer. This is one of the few renumber routines that is **non-destructive to machine language code** contained in Applesoft listings!
APPEND — INTEGER & APPLESOFT	Easily join program modules together.
ADDRESS/HEX CONVERTER	Converts all of the Apple's address formats including high- and low-order bytes for pointers.
LINE FIND — INTEGER & APPLESOFT	Find the location of any BASIC line in memory for repairing garbaged programs, or inserting HIMEM:, CLR, etc. directly into a listing.
VAL( ) & STR$( )	Instructions for simulating these Applesoft functions in Integer BASIC. Convert strings to numbers and back again with these.
SCREEN PAGE MAP	Put any character anywhere on the screen by 'POKE'ing directly into memory the proper values, given by this program.
MOVE	Move blocks of memory up or down any number of bytes from Integer BASIC or Applesoft.

The extensive documentation alone is worth the price! It includes an in-depth explanation of the internal workings of Integer BASIC and Applesoft to allow you to do things you never thought possible on your APPLE!

Southwestern Data Systems
P.O. Box 582
Santee, CA 92071
(714) 562-3670

PROGRAMMER'S UTILITY PACK

REFERENCE MANUAL

PROGRAMMER'S UTILITY PACK

Section — Page

DISCLAIMER

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

APPLE II UTILITY PACK

Description and Instructions for Program Use:

ADDRESS CONVERSION: (3.8K)

On the Apple II, an address (or any number for that matter) can be expressed in four ways. In Integer BASIC, all numbers above 32767 are expressed as a negative number. In Applesoft, any address from -65535 to +65535 can be directly stated. The number also has a hexadecimal form (base 16). Last of all, any number up to 65535 can be stored in two bytes (such as in the case of line #'s and address pointers).

To use the program, 'RUN ADDRESS CONVERSION' will do the trick. After the title banner appears, press any key to proceed. Then enter any number from -65535 to 65535 (or in hex notation, $Ø to $FFFF), and it will be displayed in all formats.

Examples: a) Suppose you wanted to know the hex address referred to by a 'CALL -936'. When the program asks the number to be evaluated, you would enter '-936'. The program will return '$FC58' as the hex address for the routine. Also given will be 252 and 88 as the two byte form (decimal) of the number, and 64600 as the actual decimal address.

b) Suppose you find the numbers '25' and '175' as the decimal numbers to place in a pair of pointers somewhere in memory as part of a BASIC program, and you would like to know the address referred to by these. When the program asks you for your number, enter a 'Control-A' only, followed by RETURN. The program will then ask for the High- and Low-order bytes. Enter 175 and 25 (assuming 175 was the high-order byte) and the program will return -20711, 44825 and $AF19 as the various forms of the number represented by those two bytes. You will notice that under High- and low-order bytes your '175' becomes '175/$AF'. This corresponds to the first two digits of the hex number $AF19. If you have the two bytes of an address in hex, it is easy to determine the final number by simply putting the two together.

SCREEN FIND: (4.3K)

Characters may be directly displayed on the Apple II's screen by 'POKE'ing the proper value into memory locations $400 to $7FF for pg. 1 and $800 to $BFF for pg. 2. This program will determine the proper address in memory and the value to enter there for the desired character.

To use the program 'RUN SCREEN FIND' in the usual manner. Then enter the horizontal and vertical position for the character just as you would normally specify a HTAB (from 1 to 40) and VTAB (from 1 to 24) statement. Then enter the character, and whether it is to be displayed in normal, inverse, or flashing mode. The program will then return the appropriate memory location, and the decimal and hex. values to put there.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

The heart of the program is on lines 440 & 450. You may find this of use in other programs of your own. There are REM statements at the end of the program to explain each variable of the equation. In general, the equation returns the mrmory address desired, given the horizontal and vertical tab positions. You may find the article, 'An Apple II Page 1 Map' by M.R. Connolly Jr. in Dec.-Jan '79 (pg. 8-41) in MICRO magazine to be of interest.

If you are interested in outputting characters to the screen from machine language, you may be interested in using the BASCALC routine in the Monitor at $FBC1. Just put the vertical position (from Ø to 39) in the Accumulator, then JSR to $FBC1. The routine will deposit the low- and high-order bytes for the first position of the horizontal line at that vertical spot in $28 and $29, respectively (40 and 41 decimal). Then add whatever the horizontal displacement is to that value. (starting at Ø for the left margin, 39 for the right).

It is beyond the scope of this documentation to go inot further detail on how to use machine language, but perhaps the above hints will be helpful.

If you would like to print on Page 2 of text from BASIC, you may wish to experiment with the program below:

```
10 LOMEM: 3072
20 DIM A$(10)
30 FOR I=2048 to 3071: POKE I,160: NEXT I
40 CALL -936
50 INPUT" H,V ON PG. 2" ,H,V
60 VTAB V: TAB H: PRINT "";
70 POKE -16299,Ø
80 POKE 41, PEEK(41) + 4
90 PRINT "THIS IS A TEST";
100 INPUT A$
110 POKE -16300,Ø
120 PRINT : GOTO 50
```

This is inspired by Andy Hertzfeld's article in the San Francisco Apple Core's Newsletter, The Cider Press, Vol.2, No. 2. Basically the idea is to let the Monitor calculate a horizontal and vertical position (#6Ø). Then change the pointer in 41 to page 2. (#8Ø). Thereafter, everything printed will appear on page 2 until the next carriage return, line feed, or VTAB. Each time you begin a new line the POKE must be repeated.

If you are unsure of how to put LOMEM: into a program, see the section on the internal structure of Integer in this documentation. Otherwise, omit this line and type it in manually before running the program. #3Ø clears pg. 2 by POKEing the value for a space into each location. #7Ø switches the display to page 2, #11Ø back to page 1.

You may wish to re-write the Monitor's scrolling routine in a location in RAM for use with page 2. Another of my programs, 'ROGER'S EASEL' makes use of this to allow the user to access help instructions at any time during the program. You might want to examine the machine language portion at the end of that program to see how page 2 can be used for this.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

RENUMBER-I: (9.4K)

This utility renumbers all line #'s from Ø to 65Ø23 at the increment of your choice in any Integer Basic program. You may renumber with new line #'s as high as 65535. The reason for the limit on numbers you can change is to protect copyright statements in other programs from being changed or deleted. You may wish to put your own statements at the end of your programs and number them at 65535 with this program.

During the renumbering process, all GOTO's, etc. are also renumbered, and you are alerted to any referenced lines not in the program. You are also alerted to any GOTO's, etc. followed by a variable or expression.

To use the program, your program should already be in memory. (LOAD it normally if it is not.) Then type 'EXEC RENUMBER-I'. The disk drive will come on and you will see a series of prompts on the screen. The title banner will then appear, at which time you can press any key to proceed.

The program will ask where in your program you would like the renumbering process to start at, what new line # to put there, what amount to increment each succeeding line, and where to end the renumbering at. If you wish to have several line numbers with the same # you may do this by giving an increment size of 'Ø'. (Such as for copyright statements at the end of programs)

You may default on these inputs with the following results: Pressing 'RETURN' only for the starting point will default to 'Ø'. The new number to be put there defaults to '10' as does increment size. The line to end on will default to 65Ø23. To renumber an entire program, starting with '1Ø' and incrementing by '1Ø's, just press 'RETURN' alone in response to each question.

Because the greates amount of time spent by the routine is searching for GOTO's, etc. if you are renumbering only REMark statements you may elect not to do this search and renumber line #'s only. This question when asked defaults to searching for GOTO's.

If for some reason (such as hitting 'RESET' or 'Control-C') the program is halted, you may recover your original program (partly renumbered) with a 'GOTO 57Ø'. You may re-activate the routine with a 'CALL RN' or a CALL to the number given when the program terminated. This is of course, providing you don't change too many lines in your program so as to over-write the location of the renumbering program. (About 500 bytes below your program)

The renumbering takes about 40 seconds per 1K of your program to do the complete renumbering.

RENUMBER-I(S): (5.2K)

This program is a shorter version of the above, intended for use in machines with 16 or 32K of memory. It will NOT work in a 48K machine. Otherwise, it is identical in function and use to the above program.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

RENUMBER-A/S: (5.3K)

'EXEC RENUMBER-A/S' to use this program. The main difference between this and the Integer version (besides the obvious fact that this is intended for use with Applesoft programs) are first, that the limit to line numbers you can change is 65ØØØ. Some other features include a display of the lines it is currently working on (more entertaining than watching a blank screen...) and more sophistication in checking for renumbering problems, such as new numbers going out of range, or an overlap or conflict when you are renumbering just portions of a program. If you halt the program with a 'Control-C' you will immediately recover your program. (again, partly re-numbered). If you should happen to hit 'RESET' you may recover your program with a 'Control-C' followed by 'GOTO 76Ø'.

You may default on any question by pressing 'RETURN' alone. Renumbering takes about 1Ø seconds for each 1K of your program length.

LINE FIND-I: (3.9K)

'EXEC LINE FIND-I' to use. Enter the line number you're looking for and this routine will return the decimal and hex. addresses of that line's location in memory. Your program should already be in memory before using this routine. Later sections in this documentation (Internal structure of Integer / Applesoft) more fully explain varius uses of this and the Apple-soft version of these programs.

LINE FIND-A/S: (3.2K)

'EXEC LINE FIND-A/S' to use. The same in function and use as the above program, but for use with Applesoft programs.

APPEND-A/S: (ØK)

This routine will allow you to add one Applesoft program directly to the end of another. This is especially useful for adding copyrights and often used sub-routines to the end of programs. To use, normally load your first program (to be at the beginning of the final version) if it is not already in memory. Then set A$ = 'NAME' where NAME is the name of the other program to be added onto the end of the first program. This second program must already be on the same disk as the APPEND-A/S routine. Then type 'EXEC APPEND-A/S' and the program on the disk will be appended to the one currently in memory.

If you have too many programs to be appended to be put on the utility disk you may use the APPEND FILE CREATE program which will put the APPEND 'EXEC' file on any disk you wish. The program is self-documenting.

APPEND-I: (ØK)

This is very similar to the above routine. However, since Integer is structured a little differently, programs appended to ones already in memory will appear at the beginning of the final listing, as opposed to at the end as in Applesoft. You must also remember to dimension A$ before setting it equal to the name of the program to be appended. 'DIM A$(4Ø)' in the immediate mode will be sufficient. APPEND FILE CREATE will also create the Integer Append file on any disk of your choice.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

VAL:

This is included as a demonstration of how to simulate Applesoft's VAL( ) function in Integer. You can use lines 32005 and 32010 as a subroutine, or put them directly in the program where you need the VAL( ) function at that point. VAL( ) converts a string to a number. See the Applesoft Reference Manual, pg. 59, for further details.

STR$:

Similar to above, but this time a simulation of Applesoft's STR$( ) function. (Converts a number to a string.)

MOVE:

A machine language routine to move blocks of memory up or down, by any increment you wish, even one byte. There is a routine in the Monitor similar to this, but it will only move blocks down if you wish to move only a few bytes. (It will work in either direction providing the distance moved is greater than the length of the block). The other drawback is that the Monitor routine is difficult to execute from Applesoft. The routine on this Utility Pack can be used from either Integer or Applesoft. It is useful, for example, for moving text or graphics (High- or Low-res.) from pg. 1 to pg. 2 or back. See the green Applesoft Manual, pg. 126 for appropriate addresses for this. Also, the Sept. 1978 issue of CONTACT, the Apple II newsletter discusses this. The machine language routine resides from $300 to $33F.

To use the MOVE routine from within a program, some pointers must be set for the beginning and end of the block you wish to move. In general, these are:

(1) POKE 60, old start addr. MOD 256
POKE 61, old start addr. / 256

(2) POKE 62, old end addr. MOD 256
POKE 63, old end addr. / 256

(3) POKE 64, new end addr. MOD 256
POKE 65, new end addr. / 256

(4) POKE 66, new start addr. MOD 256
POKE 67, new start addr. / 256

To execute the move: (assuming MOVE has already been 'BLOAD'ed)

'CALL 768' for a move up (#4 not needed for this)
'CALL 817' for a move down (#3 not needed for this)

Since Applesoft does not have the MOD and '/' functions of Integer, you will need to make use of the following relationships:

X MOD Y (in Integer) is equivalent to... X - INT(X/Y)*Y (in Applesoft)
X / Y is equivalent to... INT(X/Y)

UPDATES AND CORRECTIONS: Be sure to mail your reply card so as to be notified of any changes or additions on these programs. Any comments or suggestions you may have as to improvements in the programs is also appreciated. Please include them on the response card, or call or write the above address.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

LINE FIND - I (AND RELATED TOPICS...)

LINE FIND-I is designed to allow you to manipulate program lines at the machine level. Before going into this however, a brief discussion of the way INTEGER Basic program lines are formatted is in order.

INTERNAL STRUCTURE OF INTEGER BASIC

First of all, INTEGER stores the values for the beginning and ending points of any program in certain locations in memory called 'pointers'. Pointers are almost always located in hex addresses $ØØ to $FF (Ø to 255 decimal). This area of memory is often call the "zero page" of memory. Depending on what language you're using (Monitor, INTEGER or APPLESOFT), certain addresses are always used for a given pointer. In INTEGER, for instance, hex $CA and $CB (dec. 2Ø2 and 2Ø3) hold the starting address of the program currently in memory. If you are in Monitor and you type in 'CA CB (CR)' you will get something like this:

```
CA- F5
CB- BF
```

This tells you that stored in location $CA is the value $F5 and in $CB is $BF. These two addresses taken together tell you the starting address of the program currently in memory is $BFF5. (Note that the leading 2 digits are reversed with the last two digits as far as how they are stored in $CA and #CB. This is what is meant by the terms "high- and low-order bytes". It takes two bytes to store an address in the Apple under most circumstances. The first part of the address ($BF in our example) is called the high-order byte because it represents the larger part of the number. (Just as in the decimal system, for the number '15', the '1' represents a greater value than the '5'.) The second part of the address ($F5) is called the low-order byte. In the pointers, the low-order byte comes first followed by the high-order byte.

The end of a program is similarly stored in the zero-page of memory. For INTEGER, this is at hex $4C and $4D. (dec. 76 and 77) In the Monitor, typing in '4C 4D (CR)' might get you something like this:

```
4C- ØØ
4D- CØ
```

This would indicate that the program ended at the address $CØØØ.

Now, how are the individual lines formatted? The best way to find this out is by example, aided by the page included in the documentation. This page has "INTEGER TOKENS" in the upper left corner.

For starters, make sure your Apple is properly initialized by hitting 'RESET', followed by a 'Control-B' to get into INTEGER Basic. Now type in this simple program:

```
1Ø GOTO 2Ø
2Ø END
```

LIST and check to make sure it is identical to the example. Now to examine this at the machine level, type in 'CALL -151' or hit 'RESET' to get into the Monitor. Now find the beginning and end of the program by looking at the pointers at $CA,CB,4C and 4D.

Do this by typing in 'CA CB 4C 4D (CR)'. This should yield:

```
ØØCA- F3
ØØCB- BF
Ø04C- ØØ
Ø04D- CØ
```

(This is for a 48K machine. If you have a 16K or 32K the numbers at $CB and $4C will change accordingly...)

This tells us the program resides from $BFF3 to $CØØØ in memory. The program is short enough we can examine its entirety by typing in: 'BFF3.CØØØ (CR)' This should give:

```
BFF3- Ø8 ØA ØØ 5F B2
BFF8- 14 ØØ Ø1 Ø5 14 ØØ 51 Ø1
CØØØ- ØD
```

(Do not be concerned if the value at $CØØØ is not 'ØD'. This is not actually part of your program, but the very next byte after it...)

Referring to the two pages of INTEGER tokens, we can decode this data to see how the program is stored. The reason they are called 'tokens' is because often-used parts of a program, namely the commands and statements, are encoded in a single number for the entire word. For example, everywhere you have the command 'GOSUB' in a program, the computer stores this as a hexadecimal number, $5C. This provides a great space savings by using this technique. A few other fundamentals: 1) Every line begins with one byte that gives the length of the tokenized line. 2) The next two bytes store the line # in low- and high-order bytes. In fact, every constant or line # in INTEGER is stored using the low- and high-order bytes. 3) Each line has a 'Ø1' at the end of the line to indicate the end.

Let's look at the program itself. At $BFF3, the first byte is 'Ø8'. This tell the computer (and us) that the line is a total of eight bytes long. The next two bytes, 'ØA' and 'ØØ' are the low- and high-order bytes for the line #, in this case '1Ø'. You may use the program "ADDRESS/HEX CONVERTER" to determine the low- and high-order bytes for any number. The next byte, '5F' is the token for 'GOTO'. (Check 3rd column, first pg. to confirm) If you had typed 'GOSUB 2Ø' this would have been '5C'. Now a bit of a peculiarity. The next byte is 'B2'. This is the ASCII value for the digit '2', the first digit of the number following the 'GOTO' in your program. If you had written 'GOTO 5ØØØ' this byte would have been 'B5', the ASCII value for '5'. The following two bytes store the number '2Ø' in the low- and high-order bytes '14' and 'ØØ'. Every constant and referenced line number therefore takes a total of 3 bytes to store it in INTEGER Basic. The last byte of the line is the 'Ø1' at $BFFA. This indicates the end of the first line.

The format continues in a similar pattern for the next line. 'Ø5' shows the line is 5 bytes long. '14' and 'ØØ' encode the line number, in this case '2Ø'. '51' is the token for 'END' and 'Ø1' is the end of the line, and in this case, the end of the program. Note that the pointer at $4C,4D always points at the <u>next</u> byte after the end of the program.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

SOME USES OF THE LINE FIND PROGRAM

First, an experiment. Use Control-B to re-enter INTEGER Basic and type in 'NEW'. Suppose you were writing a program, and you wanted to clear the variables to 'Ø' at some point. Simple enough, use the CLR command, right? Not quite. Type in the following line: '1Ø CLR'

Having difficulty? This is because INTEGER Basic does not directly accept CLR as a legal syntax when typing in your line. This does not, however, mean that you are out of luck. There is a way!

Type in this: '1Ø REM' Now find out where your program is at in memory by looking at the pointers as discussed earlier. Type in 'CALL -151' or hit 'RESET' to enter the Monitor. Now enter 'CA CB 4C 4D (CR)'

This will yield the beginning and ending address of the program. (And in this case of the line itself.) List out the bytes by typing in the starting address, a period ('.'), and then the ending address. (Remember that the low- and high-order bytes are reversed when looking at $CA,CB and $4C,4D.) On a 48K sytem this would yield:

BFFB- Ø5 ØA ØØ Ø1
CØØØ- ØD

'5D' is the token for the 'REM' part of the line. What we need to do is to change this to 'ØC' for 'CLR'. There are two ways to do this. The first is to recognize that 'BFFB' is the address of the first byte shown on that line, namely the 'Ø5'. Counting over, we see that the '5D' is at hex address $BFFE. Now, change the contents of this location by typing in 'BFFF: ØC (CR)'.

If you are uncomfortable with the hex notation, there is a second way. Using the 'ESCAPE' and 'D'key, move up until the cursor is on the same line as the 'BFFB'. (It should probably be on the first 'F' in 'BFFB') Now use the 'ESCAPE' and 'B' key to move to the left one space. The cursor should now be on the 'B'. Now use the right-arrow key to copy over the addrss. When the cursor is over the hyphen ('-') enter a colon (':'). We have just simulated typing in 'BFFB: '. Now use the right-arrow key again to copy over everything on the line. Stop when you get to the '5D'. When the cursor is over the '5' in '5D' type in 'ØC' and press 'RETURN'.

Using either method we should now have an 'ØC' where the '5D' used to be. You may want to list it to check by typing 'BFFB.CØØØ (CR)'.

Now 'Control-C' to Basic and LIST. The program should now be:

1Ø CLR

Just what we wanted! HIMEM: and LOMEM: can also be entered in a similar fashion. The main restraint to replacing tokens is that whatever you type in for the temporary values must take up the same number of tokens as what you are going to replace it with. The new tokens must also still have the correct syntax, and since you are defeating having the computer check the syntax you must do this yourself. You must make sure that the new set of tokens taken as a whole for that line are consistent with what the Apple expects to find there.

For HIMEM: and LOMEM: 'PR#' may be used as your temporary statement. For instance, to end up with a line with 'HIMEM: 8192' in it, first type 'PR# 8192' in the appropriate spot. Then replace the '7E' token for PR# to a 'Ø' or '1Ø' for 'HIMEM:'.

After all this, where does LINE FIND fit in? In our examples, the line was easy to find because the programs were very short. However, for a long program it would be very tedious to try to find a line in the middle.

Solution? Just use the LINE FIND program, telling it what line # you're looking for. It will find the line, give you the hex starting and ending addresses of that line, and leave you in Monitor to list out the bytes. Just type in the starting address, a period, and then the ending address. Now you can alter anything you wish.

RECOVERING GARBAGED PROGRAMS

Occassionally during transfer of a program, or because of a faulty RAM chip, a program will drop a bit somewhere. If the alteration is in the middle of a line, where you had a 'PRINT' will suddenly become 'HIMEM:' or some similar item. You should now be alble to understand how a random error can produce a seemingly intelligent series of letters. What was changed was the single token for 'PRINT', it now being interpreted as 'HIMEM' or whatever. This is usually easily fixed however by just retyping the line.

The problem comes when the alteration occurs in a more crucial place. Namely, one of the first three or the last byte. If the first byte is changed, the program will LIST out o.k., but 'hang' on this line when you attempt a 'RUN'. If either the second or third bytes are changed the line # of that line will be altered, possibly disturbing the logic of execution of the program, or defeating GOTO's referenced past that line. If the last byte is changed, the two lines on either side of the 'Ø1' become merged, usually with the second line listing as gibberish. (The effect of having the syntax of tokens disturbed) In fact, the alteration of this one byte can make enough of a difference to have the entire program from that point on list as nonsense.

LINE FIND can be used on occasion to recover damaged programs because you can look directly at the part of memory where the problem is. There is no standard method for repair, but in general follow this procedure:

First determine which of the four types of problems above have occurred. If you have just a bad byte within the line retype it. If the line # has been changed, several things may happen. If the line # is lower than the other line #'s preceeding it, you will not be able to delete it. If it equals a line # already there before it, attempting to delete it would only delete the other line. In general, the solution is to use the LINE FIND program to find the line just ahead of the damaged one. Then go in and look at the next 3 bytes after the 'Ø1' at end of the one you found. Bytes 2 and 3 are the altered line # bytes. Chances are only one of these two has been altered. Restore these two to the low- and high-order bytes for the line # you want there and your problem should be solved. (Use ADDRESS/HEX CONVERTER if you are unsure what the two bytes should be for the line #)

If the system 'hangs' on a given line, use the LINE FIND program to find that line number. The ending address that it gives you will not be correct but the starting address will be. Then examine the part of memory after the starting address (for up to 255 bytes) to try to determine where that line actually ends. Things to look for are the 'Ø1' at the end of the line (although this may occur as part of a constant or referenced line # as discussed earlier) or use the ADDRESS/HEX CONVERTER to determine the low- and high-order bytes for the number itself of the next line, and look for these. When you have determined where the line ends, you must then determine its length and put this value back into the first byte. You may find the built-in hex subtraction routine in Monitor helpful. Subtract the starting address (in hex) from the ending address to determine the length.

If the problem is merged lines then the last bit of the line has been changed from 'Ø1' to something else. Use the LINE FIND program to find where the line was supposed to end and put an 'Ø1' there. This should do it.

These techniques should solve many of the problems encountered in damaged programs. It may seem a bit involved, but it is easier than retyping an entire program - or not having one at all if it's one you bought!. It does take a little effort and careful thought, but can be very educational in the process. Good luck!

Both LINE FIND-I and LINE FIND-A/S are good for learning how lines in either language are tokenized. To experiment, type in a line, then run LINE FIND. After listing out the line in Monitor, use 'Control-C' to return to Basic and LIST the line to compare the two. Then make any changes you wish and execute the appropriate 'CALL ...' to re-activate LINE FIND.

You can do this over and over again to see just how each kind of line is formatted.

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

INTEGER TOKENS

Dec.	Hex.	Char.	Dec.	Hex.	Char.	Dec.	Hex.	Char.
Ø	Ø	HIMEM:	43	2B	!	85	55	FOR
1	1	(End of Line)	44	2C	!	86	56	=
2	2	_ (Underscore)	45	2D	(	87	57	TO
3	3	: (Stmt sep.)	46	2E	PEEK	88	58	STEP
4	4	LOAD	47	2F	RND	89	59	NEXT
5	5	SAVE	48	3Ø	SGN	9Ø	5A	,
6	6	CON	49	31	ABS	91	5B	RETURN
7	7	RUN	5Ø	32	PDL	92	5C	GOSUB
8	8	RUN	51	33	RNDX (?)	93	5D	REM
9	9	DEL	52	34	(	94	5E	LET
1Ø	A	, (for DEL)	53	35	+	95	5F	GOTO
11	B	NEW	54	36	- (Signs)	96	6Ø	IF
12	C	CLR	55	37	NOT	97	61	PRINT (" ")
13	D	AUTO	56	38	(	98	62	PRINT (X,X$)
14	E	, (for AUTO)	57	39	=	99	63	PRINT (Null)
15	F	MAN	58	3A	#	1ØØ	64	POKE
16	1Ø	HIMEM:	59	3B	LEN(	1Ø1	65	,
17	11	LOMEM:	6Ø	3C	ASC(	1Ø2	66	COLOR=
18	12	+	61	3D	SCRN(	1Ø3	67	PLOT
19	13	-	62	3E	,	1Ø4	68	,
2Ø	14	*	63	3F	(	1Ø5	69	HLIN
21	15	/	64	4Ø	$ (String)	1Ø6	6A	,
22	16	=	65	41	$	1Ø7	6B	AT
23	17	#	66	42	(	1Ø8	6C	VLIN
24	18	>=	67	43	,	1Ø9	6D	,
25	19	>	68	44	,	11Ø	6E	AT
26	1A	<=	69	45	,	111	6F	VTAB
27	1B	<>	7Ø	46	,	112	7Ø	= (String)
28	1C	<	71	47	,	113	71	=
29	1D	AND	72	48	,	114	72	)
3Ø	1E	OR	73	49	,	115	73	)
31	1F	MOD	74	4A	,	116	74	LIST (From-to)
32	2Ø	^	75	4B	TEXT	117	75	,
33	21	+	76	4C	GR	118	76	LIST (Entire prog.)
34	22	(	77	4D	CALL	119	77	POP
35	23	,	78	4E	DIM (String)	12Ø	78	NODSP (String)
36	24	THEN (line #)	79	4F	DIM (Variable)	121	79	NODSP (Var.)
37	25	THEN (Stmnt)	8Ø	5Ø	TAB	122	7A	NOTRACE
38	26	, (String)	81	51	END	123	7B	DSP (String)
39	27	, (Variable)	82	52	INPUT (String)	124	7C	DSP (Var.)
4Ø	28	" (Beginning)	83	53	INPUT ($ or Var. w/ " ",)	125	7D	TRACE
41	29	" (Ending)				126	7E	PR#
42	2A	(	84	54	INPUT (Var.)	127	7F	IN#

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

* * * * * * * * * * *
* INTEGER TOKENS *
* * * * * * * * * * *

Dec.	Hex.	Char.
128	80	Spc.^{c}
129	81	A^{c}
130	82	B^{c}
131	83	C^{c}
132	84	D^{c}
133	85	E^{c}
134	86	F^{c}
135	87	G^{c}
136	88	H^{c}
137	89	I^{c}
138	8A	J^{c}
139	8B	K^{c}
140	8C	L^{c}
141	8D	M^{c}
142	8E	N^{c}
143	8F	O^{c}
144	90	P^{c}
145	91	Q^{c}
146	92	R^{c}
147	93	S^{c}
148	94	T^{c}
149	95	U^{c}
150	96	V^{c}
151	97	W^{c}
152	98	X^{c}
153	99	Y^{c}
154	9A	Z^{c}
155	9B	Esc. or [^{c}
156	9C	\$^{c}$
157	9D	]^{c}
158	9E	^^{c}
159	9F	_^{c}
160	A0	Space
161	A1	!
162	A2	"
163	A3	#
164	A4	$
165	A5	%
166	A6	&
167	A7	'
168	A8	(
169	A9	)
170	AA	*
171	AB	+
172	AC	,
173	AD	-
174	AE	.
175	AF	/
176	B0	0
177	B1	1
178	B2	2
179	B3	3
180	B4	4
181	B5	5
182	B6	6
183	B7	7
184	B8	8
185	B9	9
186	BA	:
187	BB	;
188	BC	<
189	BD	=
190	BE	>
191	BF	?
192	C0	@
193	C1	A
194	C2	B
195	C3	C
196	C4	D
197	C5	E
198	C6	F
199	C7	G
200	C8	H
201	C9	I
202	CA	J
203	CB	K
204	CC	L
205	CD	M
206	CE	N
207	CF	O
208	D0	P
209	D1	Q
210	D2	R
211	D3	S
212	D4	T
213	D5	U
214	D6	V
215	D7	W
216	D8	X
217	D9	Y
218	DA	Z
219	DB	[
220	DC	\
221	DD	]
222	DE	^
223	DF	_
224	E0	Space
225	E1	! (a)
226	E2	" (b)
227	E3	# (c)
228	E4	$ (d)
229	E5	% (e)
230	E6	& (f)
231	E7	' (g)
232	E8	((h)
233	E9	) (i)
234	EA	* (j)
235	EB	+ (k)
236	EC	, (l)
237	ED	- (m)
238	EE	. (n)
239	EF	/ (o)
240	F0	0 (p)
241	F1	1 (q)
242	F2	2 (r)
243	F3	3 (s)
244	F4	4 (t)
245	F5	5 (u)
246	F6	6 (v)
247	F7	7 (w)
248	F8	8 (x)
249	F9	9 (y)
250	FA	: (z)
251	FB	; {
252	FC	< |
253	FD	= }
254	FE	> ~
255	FF	?

(Printer-lwr.case)

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

INTERNAL STRUCTURE OF APPLESOFT

This section will deal with how Applesoft is formatted as opposed to INTEGER. It will be assumed that you have already read the material on INTEGER, or are fairly familiar with how INTEGER is set up. For a list of Applesoft Tokens, see the green reference manual, pgs. 138,139 and 121. The ASCII values given on 138,139 are the same as the tokens used to encode these characters in memory.

The first difference between the two languages is the location of the beginning and end of program pointers. In Applesoft, the beginning is held in $67 and $68 (dec. 103,104) and the end in $AF and $B0 (dec. 175,176). Get Applesoft up and running, then type 'NEW'. Now type in the following program:

```
'10 GOTO 20'
'20 END'
```

LIST to make sure you have done this correctly. Now type in 'CALL -151' to get into Monitor. Find the beginning and end of the program by typing '67 68 AF B0' (carriage return will be assumed from now on unless otherwise noted). On a 48K system this will give:

```
0067- 01
0068- 08
00AF- 12
00B0- 08
```

(Note: If you have Applesoft in RAM the starting address will probably be $3001 and end at $3012)

List out this range with '801.812'. This should give:

```
0801- 09 08 0A 00 AB 32 30
0808- 00 0F 08 14 00 80 00 00
0810- 00 00 08
```

(The last two bytes at $811 and $812 may be different...)

The first two bytes of the line are an index to the address of the next line. In this case they point to $0809. The next two bytes contain the line # ('10'). 'AB' is the token for 'GOTO'. In Applesoft, constants and referenced line #'s are stored with one byte per digit. In our example, the number '20' is stored as '2','0'. The end of the line is indicated with a '00'. The next line continues in a similar fashion.

'0F 08' indicates that the next line would start at $80F. '14' and '00' are the low- and high-order bytes for the line number '20'. '80' is the token for 'END' and '00' indicates the end of the line. In Applesoft, the end of the line is always easy to spot because it is the only time '0' is used. Applesoft knows where the end of the program is, not by using the pointer at $AF,B0, but by finding '00 00' when it looks for the index at the end of the program. (In this case at $80F and $810.)

$AF and $B0 must point to an address no smaller than these two zeros, but it may point to any address after this.

(Note: you may find it helpful to use the ADDRESS/HEX CONVERTER to determine the hexadecimal values for the tokens on pg. 121.)

Southwestern Data Systems

P.O. Box 582
Santee, Ca. 92071
562-3670

The pointers at $AF and $BØ are used when LOADing and SAVEing a program, but not during a 'RUN'. In fact, this is very useful because you can store any kind of binary data you wish (machine language programs, screen images, etc.) within your program space by just pointing $AF and $BØ at a suitable location beyond the true end of the program and storing your binary data in the created space. You may SAVE and LOAD it just as you would any program and the binary data will be carried along right with it.

Since Applesoft does not check syntax until it RUNs a program, inserting what you want in a program line is not a prblem, but recovering garbaged programs still is. In Applesoft, if the 1st or 2nd byte is altered, strange things may happen. Lines may merge or repeat indefinitely or just disappear. Usually what happens is that the listing becomes nonsense after a certain point. The next two determine the line number, and if these are changed it is similar to what can happen with INTEGER. If the last byte is changed, the program will LIST properly, but during a 'RUN' you will get a 'SYNTAX ERROR' on that line for no apparent reason, stopping execution.

The procedure for fixing these problems is similar to that in INTEGER. If the first two bytes are suspected, find the last intelligent line # in the listing. Then go into Monitor and look for the 'ØØ' at the end of the line. The index in the first two bytes of the line should point to the address right after the 'ØØ'.

If the line # is altered, use LINE FIND to locate it if you cannot change it directly. If there are no lines before it with the same # you may search for it directly with the program. Otherwise, locate the line just before it, find the 'ØØ' at the end of that, and then put the correct low- and high-order bytes for the line # you want there into the 3rd and 4th bytes of the altered line. If the last byte is bad, use LINE LINE FIND to get the ending address and put a 'Ø' in the last byte of the line.

Southwestern Data Systems

* * * * * * * *
* APPLESOFT TOKENS *
* * * * * * * *

P.O. Box 582
Santee, Ca. 92071
562-3670

Dec.	Hex.	Char.
Ø	Ø	@c
1	1	Ac
2	2	Bc
3	3	Cc
4	4	Dc
5	5	Ec
6	6	Fc
7	7	Gc
8	8	Hc
9	9	Ic
1Ø	A	Jc
11	B	Kc
12	C	Lc
13	D	Mc
14	E	Nc
15	F	Oc
16	1Ø	Pc
17	11	Qc
18	12	Rc
19	13	Sc
2Ø	14	Tc
21	15	Uc
22	16	Vc
23	17	Wc
24	18	Xc
25	19	Yc
26	1A	Zc
27	1B	ESCape
28	1C	\c
29	1D	]c
3Ø	1E	^c
31	1F	_
32	2Ø	Space
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
4Ø	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	3Ø	Ø
49	31	1
5Ø	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
6Ø	3C	<
61	3D	=
62	3E	>
63	3F	?
64	4Ø	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
7Ø	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
8Ø	5Ø	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
9Ø	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	6Ø	Space
97	61	!
98	62	"
99	63	#
1ØØ	64	$
1Ø1	65	%
1Ø2	66	&
1Ø3	67	'
1Ø4	68	(
1Ø5	69	)
1Ø6	6A	*
1Ø7	6B	+
1Ø8	6C	,
1Ø9	6D	-
11Ø	6E	.
111	6F	/
112	7Ø	Ø
113	71	1
114	72	2
115	73	3
116	74	4
117	75	5
118	76	6
119	77	7
12Ø	78	8
121	79	9
122	7A	:
123	7B	;
124	7C	<
125	7D	=
126	7E	>
127	7F	?

(Printer-lwr.case)

(`)
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)
(q)
(r)
(s)
(t)
(u)
(v)
(w)
(x)
(y)
(z)
({)
(|)
(})
(~)

Southwestern Data Systems

* * * * * * * * * * * *
* APPLESOFT TOKENS *
* * * * * * * * * * * *

P.O. Box 582
Santee, Ca. 92071
562-3670

Dec.	Hex.	Char.
128	8Ø	END
129	81	FOR
130	82	NEXT
131	83	DATA
132	84	INPUT
133	85	DEL
134	86	DIM
135	87	READ
136	88	GR
137	89	TEXT
138	8A	PR#
139	8B	IN#
140	8C	CALL
141	8D	PLOT
142	8E	HLIN
143	8F	VLIN
144	9Ø	HGR2
145	91	HGR
146	92	HCOLOR=
147	93	HPLOT
148	94	DRAW
149	95	XDRAW
150	96	HTAB
151	97	HOME
152	98	ROT=
153	99	SCALE=
154	9A	SHLOAD
155	9B	TRACE
156	9C	NOTRACE
157	9D	NORMAL
158	9E	INVERSE
159	9F	FLASH
160	AØ	COLOR=
161	A1	POP
162	A2	VTAB
163	A3	HIMEM:
164	A4	LOMEM:
165	A5	ON ERR
166	A6	RESUME
167	A7	RECALL
168	A8	STORE
169	A9	SPEED=
170	AA	LET
171	AB	GOTO
172	AC	RUN
173	AD	IF
174	AE	RESTORE
175	AF	&
176	BØ	GOSUB
177	B1	RETURN
178	B2	REM
179	B3	STOP
180	B4	ON
181	B5	WAIT
182	B6	LOAD
183	B7	SAVE
184	B8	DEF
185	B9	POKE
186	BA	PRINT
187	BB	CONT
188	BC	LIST
189	BD	CLEAR
190	BE	GET
191	BF	NEW
192	CØ	TAB(
193	C1	TO
194	C2	FN
195	C3	SPC(
196	C4	THEN
197	C5	AT
198	C6	NOT
199	C7	STEP
200	C8	+
201	C9	-
202	CA	*
203	CB	/
204	CC	^
205	CD	AND
206	CE	OR
207	CF	>
208	DØ	=
209	D1	<
210	D2	SGN
211	D3	INT
212	D4	ABS
213	D5	USR
214	D6	FRE
215	D7	SCRN(
216	D8	PDL
217	D9	POS
218	DA	SQR
219	DB	RND
220	DC	LOG
221	DD	EXP
222	DE	COS
223	DF	SIN
224	EØ	TAN
225	E1	ATN
226	E2	PEEK
227	E3	LEN
228	E4	STR$
229	E5	VAL
230	E6	ASC
231	E7	CHR$
232	E8	LEFT$
233	E9	RIGHT$
234	EA	MID$
235	EB	
236	EC	
237	ED	
238	EE	
239	EF	
240	FØ	
241	F1	
242	F2	
243	F3	
244	F4	
245	F5	
246	F6	
247	F7	
248	F8	
249	F9	
250	FA	
251	FB	
252	FC	
253	FD	
254	FE	
255	FF	